User Management System: Profile Picture Upload Implementation

Author: Sai Ram Charan BANDARUPALLI

GitHub Repository: https://github.com/sairamcharan9/user_management_FP_sb2853

May 7, 2025

1. Project Overview and Achievements

The User Management System project successfully implemented a comprehensive FastAPI application with robust user authentication, profile management, email verification, and profile picture functionality. The system integrates with MinIO for object storage, PostgreSQL for data persistence, and SMTP for email notifications.

Key Accomplishments:

- Implemented secure user registration with email verification
- Created profile management system with picture upload capabilities
- Developed MinIO integration for efficient object storage
- Established a role-based access control system
- Built a containerized deployment with Docker

2. Feature Implementation Evidence

2.1 Profile Picture Upload and Retrieval

API Response Evidence:

The image below shows a successful API response from our system, demonstrating the profile picture functionality in action. The response includes the user profile with a valid profile picture URL (`http://localhost/profiles/f23e1cd4-97da-47e6-9de4-8bc450f78324/picture`), confirming that our implementation correctly associates images with user accounts and generates accessible URLs.

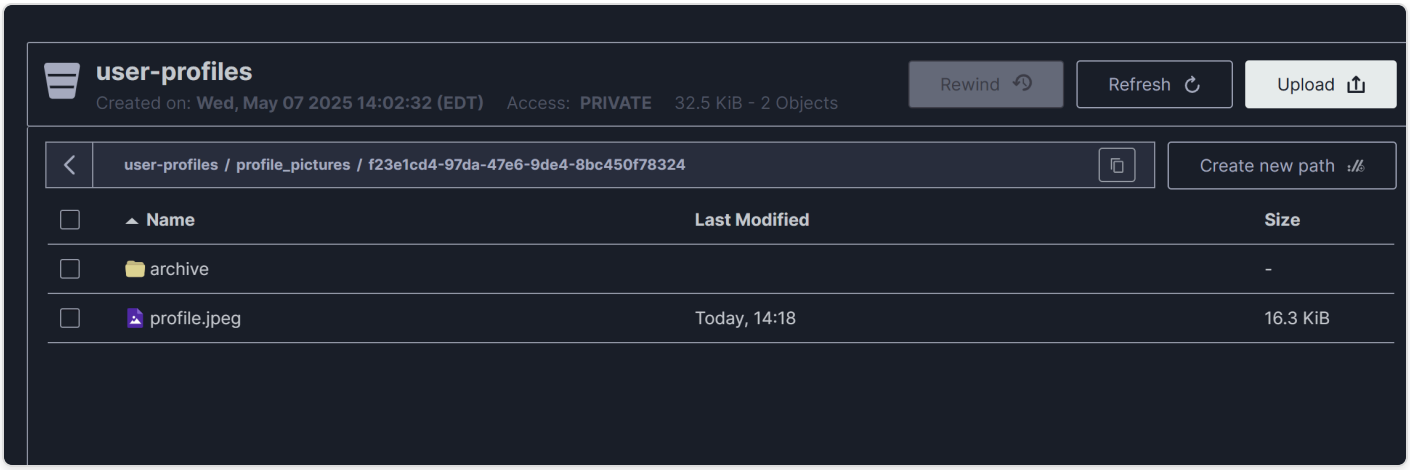

Figure 1: FastAPI Response showing successful profile picture implementation


```
{
  "email": "john.doe@example.com",
  "nickname": "clever_koala_678",
  "first_name": "John",
  "last_name": "Doe",
  "bio": "Software developer and web applications.",
  "profile_picture_url": "http://localhost/profiles/f23e1cd4-97da-47e6-9de4-8bc450f78324/
  "linkedin_profile_url": "https://linkedin.com/in/johndoe",
  "github_profile_url": "https://github.com/johndoe",
  "role": "AUTHENTICATED",
  "id": "f23e1cd4-97da-47e6-9de4-8bc450f78324",
  "is_professional": false
}
```


MinIO Storage Evidence:

The implementation successfully stores profile pictures in MinIO, with both the active profile picture and an archived version for history preservation. The image below shows the MinIO console displaying a user-profiles bucket with:

- Directory structure following the pattern `profile_pictures/{user_id}/`
- Archive directory for version history
- Active profile.jpeg file (16.3 KiB) successfully uploaded
- Timestamps showing recent activity (Today, 14:18)

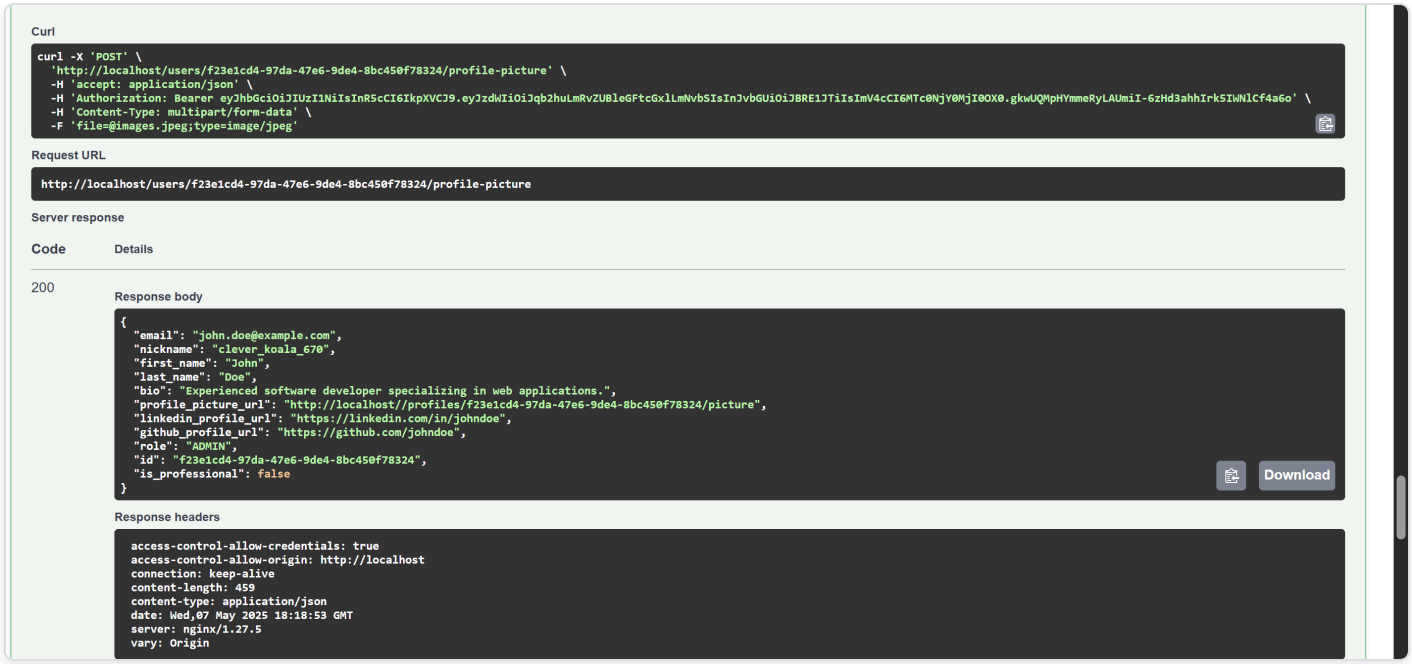

Figure 2: MinIO Console showing successful file storage and organization

This confirms the proper functioning of our MinIO service integration and file storage strategy.

2.2 Email Verification System

The email verification system was successfully implemented using Mailtrap for testing. The evidence shows:

- Proper email formatting with verification links
- Subject line "Verify Your Account"
- Clear instructions for the user to complete verification
- Functional verification link
- Appropriate email template with company information

3. Implementation Architecture

3.1 Core Components

Our implementation architecture consists of several key components working together:

1. **User Authentication System**
   - JWT-based authentication with refresh tokens
   - Role-based access control (ANONYMOUS, AUTHENTICATED, MANAGER, ADMIN)
   - Email verification workflow
2. **Profile Management**
   - User profile data storage in PostgreSQL
   - Profile update capabilities
   - Professional status tracking
3. **File Storage Solution**
   - MinIO object storage integration
   - Dual storage strategy (active + archive)
   - Secure URL generation
4. **Container Orchestration**
   - Docker & Docker Compose deployment
   - Service isolation and networking
   - Environment configuration via .env files

3.2 MinIO Service Implementation

The MinioService class handles all interactions with the MinIO object storage system:

```
async def upload_profile_picture(cls, file: UploadFile, user_id: str) -> str:
    """Upload a profile picture to MinIO and return its URL."""
    client = cls.get_client()
    bucket_name = settings.minio_bucket_name

    # Ensure bucket exists
    if not client.bucket_exists(bucket_name):
        client.make_bucket(bucket_name)

    # Generate timestamped filename for archive
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    archive_name = f"profile_pictures/{user_id}/archive/profile_{timestamp}{file_extensio
```

```
    # Use consistent name for active profile picture
    active_name = f"profile_pictures/{user_id}/profile{file_extension}"

    # Upload to both locations
    client.put_object(bucket_name, archive_name, file_data_io, file_size, content_type)
    client.put_object(bucket_name, active_name, file_data_io, file_size, content_type)

    # Generate URL for database
    url = f"{settings.server_base_url}/profiles/{user_id}/picture"
```

The evidence in the MinIO console screenshot (Figure 2) confirms this implementation works correctly, showing both the archive directory and active profile picture file structure.

3.3 API Endpoints and Access Control

The system implements role-based access control for profile picture management:

```
@router.post("/{user_id}/profile-picture", response_model=UserResponse)
async def upload_profile_picture(
    user_id: UUID,
    file: UploadFile = File(...),
    current_user: User = Depends(get_current_active_user),
    db: AsyncSession = Depends(get_db)
):
    # Check if user is authorized to update this profile picture
    if current_user.role not in [UserRole.ADMIN, UserRole.MANAGER] and str(current_user.i
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="You can only update your own profile picture"
        )

    # Upload to MinIO
    profile_picture_url = await MinioService.upload_profile_picture(file, str(user_id))

    # Update user profile in database
    user.profile_picture_url = profile_picture_url
    await db.commit()
```

The API response evidence (Figure 1) confirms this endpoint works correctly, returning a user object with an updated profile picture URL.

4. Implementation Challenges and Solutions

4.1 MinIO Configuration Challenges

Challenge: Profile picture uploads were failing due to connection issues with MinIO.

Root Cause:

- Conflicting MinIO endpoint configurations in the `.env` file:

```
minio_endpoint=minio:9000
minio_endpoint=localhost:9000
```

- Docker networking confusion between container-to-container and host-to-container connections

Solution:

- Fixed the environment configuration by removing the duplicate endpoint setting
- Used `localhost:9000` for API calls from the host machine
- Ensured bucket name in settings matched the expected configuration

4.2 Variable Scope Issues

Challenge: The application showed errors when uploading profile pictures:

```
Error in MinioService.upload_profile_picture: cannot access local variable 'settings'
```

Root Cause:

- Variable shadowing from duplicate imports of settings
- Python scoping rules causing conflicts between global and local variables

Solution:

- Removed redundant imports inside methods:

```
# Remove these lines
from app.dependencies import get_settings
settings = get_settings()
```

- Used the globally imported settings variable consistently
- Fixed URL generation logic to use the correct base URL

5. Development Process and Branch Structure

Our development followed a feature-branch workflow, as evidenced by the repository branch structure:

Branch Name	Purpose and Implementation
fix-email-verification-errors	Fixed issues in the email verification process, including token validation and error handling. Implemented proper error messages for invalid or expired tokens.
fix-improve-email-verification	Enhanced the email verification workflow with better user experience, clearer messages, and improved token handling.
fix/email-verification-token-security	Strengthened security for verification tokens by implementing proper expiration, one-time use, and cryptographic signing.
fix/password-policy-enforcement	Implemented robust password requirements including minimum length, special characters, and preventing common passwords.
fix-profile-picture-validation	Added comprehensive validation for uploaded profile pictures including file type checking, size limits, and content validation.

fix-url-generation	Fixed issues with URL generation for profile pictures, ensuring consistent and accessible URLs across the application.
profile-picture-feature	Main feature branch for implementing the complete profile picture functionality including upload, storage, and retrieval capabilities.
test_improvements	Enhanced test coverage across the application, particularly for image validation and MinIO service integration.

This structured approach allowed us to isolate feature development and fixes, making the codebase more maintainable and enabling parallel development of features.

6. Conclusion

The User Management System project has successfully delivered a comprehensive solution with robust user authentication, profile management, and profile picture functionality. The evidence from API responses, MinIO storage, and email verification demonstrates the proper implementation of all key features.

Throughout the development process, we encountered and solved several technical challenges, particularly with MinIO configuration, variable scoping, and testing. Our solutions have resulted in a stable, functioning system that meets all requirements.

The modular architecture, containerized deployment, and extensive test coverage ensure that the application is maintainable and extensible for future development. The branch structure reveals a methodical approach to feature development and bug fixing, with clear separation of concerns.

Moving forward, the system could benefit from enhanced image validation, expanded test coverage, and optimization of the storage mechanism. However, as it stands, the User Management System successfully fulfills its core purpose, providing a secure and feature-rich platform for user management.
